GRMON User's Manual

Version 1.0.5

August 2004

1 Introduction

1.1 General

GRMON is a general debug monitor for the LEON processor. It comes with both a simulator backend(tsim) and a backend for the LEON debug support unit(DSU). It includes, among others, the following functions:

- Read/write access to all LEON registers and memory
- Built-in disassembler and trace buffer management
- Downloading and execution of LEON applications
- Breakpoint and watchpoint management
- Remote connection to GNU debugger (gdb)

1.2 Supported platforms and system requirements

GRMON currently supports two platforms: linux-2.2/glibc-2.2 and windows98/NT/2K. The windows version requires that cygwin, version 1.5.5-1, or higher is installed. Cygwin can be downloaded from sources.redhat.com. Note that operation on Cygwin platforms is generally less reliable, and depends a lot on used Cygwin and Windows version.

1.3 Obtaining GRMON

The primary site for GRMON is *http://www.gaisler.com/*, where the latest version of GRMON can be ordered and evaluation versions downloaded.

1.4 Installation

GRMON can be installed anywhere on the host computer - for convenience the installation directory should be added to the search path. The commercial versions use HASP license key. .

1.5 License installation

GRMON is licensed using a HASP hardware key.

1.5.1 HASP4 hardware key

Two versions of the HASP USB harwware key are available, HASP4 M1 for node-locked licenses (blue key), and HASP4 Net5 for floating licenses (red key). Before use, a device driver for the key must be installed. The latest drivers can be found at www.ealaddin.com or www.gaisler.com. If a floating-license key is used, the HASP4 network license server also has to be installed and started. The necessary server installation documentation can be obtained from the distribution CD or from www.ealaddin.com. See *appendix A* for installation of device drivers under Windows and Linux platforms.

1.6 Problem reports

Please send problem reports or comments to *grmon@gaisler.com*.

2 Operation

2.1 Overview

GRMON can operate in two modes: standalone and attached to gdb. In standalone mode, LEON applications can be loaded and debugged using a command line interface. A number of commands are available to examine data, insert breakpoints and advance execution, etc. When attached to gdb, GRMON acts as a remote gdb target, and applications are loaded and debugged through gdb (or a gdb front-end such as DDD or Emacs GUD-mode).

2.2 Command line options

GRMON is started as follows on a command line:

grmon [*options*] [*input_files*]

The following command line options are supported by GRMON:

-**dsu**	Start grmon in DSU mode. (default)
-**sim**	Start grmon in SIM mode.
-**c** *file*	Reads commands from *file* instead of stdin.
-**i**	Force a system probe and initialise LEON memory and peripherals settings.
-ni	Do not initialise memory if '-i' is set.
-**gdb**	Listen for gdb connection directly at start-up.
-**port** *gdbport*	Set the port number for gdb communications. Default is 2222.
-**v**	Turn on verbose mode. (debug 1)
-**vv**	Turn on extra verbose mode. (debug 2)
-**ucmd** *file*	Load a user command module. (professional only)
-**xburn** *file*	Program the Xilinx FPGA using the Xilinx programming pod (parallel cable III/IV) in slave serial mode. If neither -dsu or -sim is specified, grmon will exit after programming. This feature is only available on linux and grmon has to be executed with root privileges. Thus, if this feature is desired, it is suggested setting the suid bit on the grmon executable and making root the owner. (i.e. *chown root:root grmon; chmod +s grmon*). *file* must be a valid Xilinx bitfile. Only available on Linux.
input_files	Executable files to be loaded into memory. The input file is loaded into the target memory according to the entry point for each segment. Recognized formats are elf32 and S-record.

Each backend also accepts a number of different options, which are described in 5.2.4 (page 19) for the DSU backend and 6.2.1 (page 26) for the simulator backend.

3 Interactive mode

Here follows a description on how to use grmon in interactive mode. If not mentioned otherwise, the operation is identical regardless of which backend is used.

3.1 Internal commands

GRMON dynamically loads libreadline.so if available on your host system, and uses `readline()` to enter/edit monitor commands. If libreadline.so is not found, `fgets()` is used instead (no history, poor editing capabilities and no tab-completion). Below is a description of those commands available on both backends, when used in standalone mode (See appendix B for a more detailed description, including the backend specific commands):

batch	execute a batch file of grmon commands
break	print or add breakpoint
cont	continue execution
dcache	show data cache
debug	change or show debug level
delete	delete breakpoint(s)
disassemble	disassemble memory
echo	echo string in monitor window
exit	see 'quit'
float	display FPU registers
gdb	connect to gdb debugger
go	start execution without initialisation
hbreak	print breakpoints or add hardware breakpoint (if available)
help	show available commands or usage for specific command
icache	show instruction cache
leon	show leon registers
load	load a file
mem	see 'x'(examine memory)
profile	enable/disable/show simple profiling (see also "aprof" command in simulator backend)
register	show/set integer registers
reset	reset active backend
run	reset and start execution at last load address
shell	execute a shell command
step	single step one or [n] times

symbols	show symbols or load symbols from file
target	change backend
quit	exit grmon
version	show version
watch	print or add watchpoint
wmem	write word to memory
x	examine memory

Typing a 'Ctrl-C' will interrupt a running program. Short forms of the commands are allowed, e.g **c**, **co**, or **con**, are all interpreted as **cont**. Tab completion is available for commands, text-symbols and filenames.

3.2 Running applications

To run a program, first use the **load** command to download the application and the **run** to start it. The program should be compiled with LECCS.

3.2.1 Running applications in dsu mode

```
grmon[dsu]> load samples/hello
section: .text at 0x40000000, size 14656 bytes
section: .data at 0x40003940, size 1872 bytes
total size: 16528 bytes (99.4 kbit/s)
read 71 symbols
grmon[dsu]> run

Program exited normally.
grmon[dsu]>
```

The output from the application appears on the normal LEON UARTs and thus not in the GRMON console, unless the **-u** switch was given at startup. The loaded applications should **not** be run through mkprom. Before the application is started, the complete integer and floating-point register file is cleared (filled with zeros), and the three memory configuration registers (MCFG1-3) and stack pointer are initialised to their default values. Various processor registers such as %psr and %wim are also initialised. When an application terminates, it must be reloaded on the target before it can be re-executed in order to re-initialise the data segment (.data).

3.2.2 Running applications in simulator mode

To switch to simulator mode and run the same application, issue the **target** command to switch backend and then do exactly as in dsu mode.

```
grmon[dsu]> target sim

 LEON SPARC simulator backend, version 1.0 (professional version)

 Copyright (C) 2001, Gaisler Research - all rights reserved.
using 64-bit time
serial port A on stdin/stdout
allocated 4096 K RAM memory, in 1 bank(s)
allocated 2048 K ROM memory
icache: 1 * 4 kbytes, 16 bytes/line (4 kbytes total)
```

```
dcache: 1 * 4 kbytes, 16 bytes/line (4 kbytes total)

 LEON SPARC simulator backend, version 1.0 (professional version)

 Copyright (C) 2001, Gaisler Research - all rights reserved.
using 64-bit time
grmon[sim]> load samples/hello
section: .text at 0x40000000, size 14656 bytes
section: .data at 0x40003940, size 1872 bytes
total size: 16528 bytes (306741.0 kbit/s)
read 71 symbols
grmon[sim]> run
resuming at 0x40000000
Hello world!

Program exited normally.
grmon[sim]>
```

3.3 Inserting breakpoints and watchpoints

Instruction breakpoints are inserted using the **break** or **hbreak** commands.

3.3.1 DSU

The **break** command inserts a software breakpoint (ta 1), while **hbreak** will insert a hardware breakpoint using one of the IU watchpoint registers. To debug code in read-only memories (e.g. prom), only hardware breakpoints can be used. Note that it is possible to debug any ram-based code using software breakpoints, even where traps are disabled such as in trap handlers.

3.3.2 Simulator

The simulator backend supports execution breakpoints and write data watchpoints. In standalone mode, hardware breakpoints are always used and no instrumentation of memory is made. When using the gdb interface, the gdb 'break' command normally uses software breakpoints by overwriting the breakpoint address with a 'ta 1' instruction. Hardware breakpoints can be inserted by using the gdb 'hbreak' command. Data write watchpoints are inserted using the 'watch' command. A watchpoint can only cover one word address, block watchpoints are not available.

3.4 Displaying registers

The current register window can be displayed using the **reg** command:

```
grmon[dsu]> register

          INS       LOCALS      OUTS     GLOBALS
   0:  00000000   00000000   00000000   00000000
   1:  00000000   00000000   00000000   00000000
   2:  00000000   00000000   00000000   00000000
   3:  00000000   00000000   00000000   00000000
   4:  00000000   00000000   00000000   00000000
   5:  00000000   00000000   00000000   00000000
   6:  00000000   00000000   00000000   00000000
   7:  00000000   00000000   00000000   00000000

 psr: 004010C6   wim: 00000001   tbr: 40000800   y: 00000000

 pc:  40000800  ta  0x0
 npc: 40000804  nop

grmon[dsu]>
```

Other register windows can be displayed using **reg w*n***, when ***n*** denotes the window number. Use the float command to show the FPU registers (if present).

3.5 Symbolic debug information

GRMON will automatically extract (.text) symbol information from elf-files. The symbols can be used where an address is expected:

```
grmon[dsu]> break main
grmon[dsu]> run
breakpoint 1  main (0x40001ac8)
grmon[dsu]> disassemble strlen 3

 40001e4c  808a2003   andcc  %o0, 0x3, %g0
 40001e50  12800016   bne  0x40001ea8
 40001e54  96100008   mov  %o0, %o3

grmon[dsu]>
```

The **symbols** command can be used to display all symbols, or to read in symbols from an alternate (elf) file:

```
grmon[dsu]> symbols samples/hello
read 71 symbols
grmon[dsu]> symbols
0x40000000 L _trap_table
0x40000000 L start
0x4000102c L _window_overflow
0x40001084 L _window_underflow
0x400010dc L _fpdis
0x400011a4 T _flush_windows
0x400011a4 T _start
0x40001218 L fstat
0x40001220 L isatty
0x40001228 L getpid
0x40001230 L kill
```

```
0x40001238 L _exit
0x40001244 L lseek
...
```

Reading symbols from alternate files is necessary when debugging self-extracting applications, such as boot-proms created with mkprom or linux/uClinux.

3.6 Displaying memory contents

Any memory loaction can be displayed using the **x** command. If a third argument is provided, that is interpreted as the number of bytes to display. Text symbols can be used instead of a numeric address.

```
grmon[dsu]> x 0x40000000

 40000000  a0100000  29100004  81c52000  01000000    ....)..... .....
 40000010  91d02000  01000000  01000000  01000000    .. ............
 40000020  91d02000  01000000  01000000  01000000    .. ............
 40000030  91d02000  01000000  01000000  01000000    .. ............

grmon[dsu]> x 0x40000000 32

 40000000  a0100000  29100004  81c52000  01000000    ....)..... .....
 40000010  91d02000  01000000  01000000  01000000    .. ............

grmon[dsu]> x main

 40001AC8  1110000e  90122010  8213c000  400000d7    ...... .....@...
 40001AD8  9e104000  01000000  9de3bf98  9e20001b    ..@.......... ..
 40001AE8  98102000  90100018  92100019  9a200019    .. .......... ..
 40001AF8  82200018  9410001a  9610001b  a620001a    . ........... ..

grmon[dsu]>
```

3.7 Disassembly of memory

Any memory location can be disassembled using the **disassemble** command:

```
grmon[sim]> disassemble 0x40000000 5

 40000000  a0100000   clr       %l0
 40000004  29100004   sethi     %hi(0x40001000), %l4
 40000008  81c52000   jmp       %l4
 4000000c  01000000   nop
 40000010  91d02000   ta        0x0

grmon[sim]> disassemble 0x40000000 0x4000000c

 40000000  a0100000   clr       %l0
 40000004  29100004   sethi     %hi(0x40001000), %l4
 40000008  81c52000   jmp       %l4
 4000000c  01000000   nop

grmon[sim]> disassemble main 3

 40001ac8  1110000e   sethi     %hi(0x40003800), %o0
 40001acc  90122010   or        %o0, 0x10, %o0
 40001ad0  8213c000   or        %o7, %g1
```

```
grmon[sim]>
```

Note that, in dsu mode, also the contents of the instruction cache can be disassembled:

```
grmon[dsu]>  disassemble 0x90140000 5

 90140000  a0100000   clr  %l0
 90140004  29100004   sethi  %hi(0x40001000), %l4
 90140008  81c52000   jmp  %l4
 9014000c  01000000   nop
 90140010  91d02000   ta  0x0

grmon[dsu]>
```

3.8 Loadable command module

It is possible for the user to add commands to grmon by creating a loadable command module. The module should export a pointer to a UserCmd_T called UserCommands, e.g.:

```
UserCmd_T *UserCommands = &CommandExtension;
```

UserCmd_T is defined as:

```
typedef struct
{
/* Functions exported by grmon */
   int  (*MemoryRead  )(unsigned int addr, unsigned char *data, unsigned int length);
   int  (*MemoryWrite )(unsigned int addr, unsigned char *data, unsigned int length);
   void (*GetRegisters)(unsigned int registers[]);
   void (*SetRegisters)(unsigned int registers[]);
   void (*dprint)(char *string);

/* Functions provided by user */
   int  (*Init)();
   int  (*Exit)();
   int  (*CommandParser)(int argc, char *argv[]);
   char **Commands;
   int    NumCommands;
} UserCmd_T;
```

The first five entries is function pointers that are provided by grmon when loading the module. The other entries has to be implemented by the user. This is how:

- *Init* and *Exit* are called when entering and leaving a grmon target.
- *CommandParser* are called from grmon before any internal parsing is done. This means that you can override internal grmon commands. On success CommandParser should return 0 and on error the return value should be > 200. On error grmon will print out the error number for diagnostics. *argv[0]* is the command itself and *argc* is the number of tokens, including the command, that is supplied.
- *Commands* should be a list of available commands. (used for command completion)
- *NumCommands* should be the number of entries in *Commands*. It is crusial that this number matches the number of entries in *Commands*. If *NumCommands* is set to 0(zero), no command completion will be done.

A simple example of a command module is supplied with the professional version of grmon.

3.9 Simple Profiling

Grmon has support for a simple form of profiling profiling in both dsu and simulator mode. In simulator mode a more sophisticated version of profiling is also available, see "Profiling" on page 29.
The simple version of profiling merely collects information of in which function the program currently executes. It does not take into consideration if the current function is called from within another procedure. Nether the less, having this in mind is could provide some useful information.

```
grmon[sim]> profile 1
Profiling enabled
grmon[sim]> run
resuming at 0x40000000
Starting
    Perm  Towers  Queens   Intmm      Mm  Puzzle   Quick  Bubble    Tree     FFT
      50      33      17     116    1100     217      33      34     266     934

Nonfloating point composite is        126

Floating point composite is        862

Program exited normally.
grmon[sim]> prof
function                 samples     ratio(%)
__unpack_f                 23627      16.92
__mulsf3                   22673      16.24
__pack_f                   17051      12.21
__divdi3                   14162      10.14
.umul                       8912       6.38
Fit                         7594       5.44
__muldi3                    6453       4.62
_window_overflow            3779       2.70
Insert                      3392       2.42
__addsf3                    3327       2.38
_window_underflow           2734       1.95
__subsf3                    2409       1.72
Fft                         2207       1.58
start                       2165       1.55
Innerproduct                2014       1.44
Bubble                      1767       1.26
rInnerproduct               1443       1.03
Place                       1371       0.98
Remove                      1335       0.95
Try                         1275       0.91
Permute                     1125       0.80
Quicksort                    995       0.71
.div                         841       0.60
Push                         657       0.47

--- a lot more output ---

grmon[sim]> profile 1
```

See 6.2.5 (page 29) for a comparison with the more sophisticated profiling only available in the simulator backend.

4 GDB interface

4.1 Attaching to gdb

GRMON can act as a remote target for gdb, allowing symbolic debugging of target applications. To initiate gdb communications, start the monitor with the **-gdb** switch or use the GRMON **gdb** command:

```
john@pluto:tmp/grmon% grmon -gdb

 GRMON - The LEON multi purpose monitor v1.0

 Copyright (C) 2004, Gaisler Research - all rights reserved.
 For latest updates, go to http://www.gaisler.com/
 Comments or bug-reports to grmon@gaisler.com

 LEON DSU Monitor backend 1.0  (professional version)

 Copyright (C) 2003, Gaisler Research - all rights reserved.
 Comments or bug-reports to jiri@gaisler.com

 using port /dev/ttyS0 @ 115200 baud

 processor frequency   :  99.53 MHz
 register windows      :  8
 v8 hardware mul/div   :  yes
 floating-point unit   :  meiko
 instruction cache     :  1 * 8 kbytes, 32 bytes/line (8 kbytes total)
 data cache            :  1 * 8 kbytes, 32 bytes/line (8 kbytes total)
 hardware breakpoints  :  4
 trace buffer          :  256 lines, mixed cpu/ahb tracing
 stack pointer         :  0x400ffff0
gdb interface: using port 2222
```

Then, start gdb in a different window and connect to GRMON using the extended-remote protocol:

```
(gdb) target extended-remote pluto:2222
Remote debugging using pluto:2222
0x40000800 in start ()
(gdb)
```

While attached, normal GRMON commands can be executed using the gdb **monitor** command. Output from the GRMON commands, such as the trace buffer history is then displayed in the gdb console:

```
(gdb) monitor hist
   time      address   instruction                result
    4484188  40001e90  add  %g2, %o2, %g3          [6e1f766e]
    4484194  40001e94  andn  %g3, %g2, %g2         [001f0000]
    4484195  40001e98  andcc  %g2, %o0, %g0        [00000000]
    4484196  40001e9c  be,a  0x40001e8c            [40001e3c]
    4484197  40001ea0  add  %o1, 4, %o1            [40003818]
    4484198  40001e8c  ld  [%o1], %g2              [726c6421]
    4484200  40001e90  add  %g2, %o2, %g3          [716b6320]
    4484201  40001e94  andn  %g3, %g2, %g2         [01030300]
    4484202  40001e98  andcc  %g2, %o0, %g0        [00000000]
    4484203  40001e9c  be,a  0x40001e8c            [40001e3c]
```

It is also possible to switch backend within a gdb session by sending the **target** command to grmon. All breakpoints will remain. However, once you switch target you need to reload the program and restart the application. The state is **not** saved during monitor target switch!

```
john@pluto:tmp/grmon-0.1% sparc-rtems-gdb
GNU gdb 5.3
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "--host=i686-pc-linux-gnu --target=sparc-tsim-elf".
(gdb) file samples/hello
Reading symbols from samples/hello...done.
(gdb) target extended-remote pluto:2222
Remote debugging using pluto:2222
0x40001ea0 in strlen ()
(gdb) lo
Loading section .text, size 0x3940 lma 0x40000000
Loading section .data, size 0x750 lma 0x40003940
Start address 0x40000000, load size 16528
Transfer rate: 66112 bits/sec, 275 bytes/write.
(gdb) break main
Breakpoint 1 at 0x40001ac8: file hello.c, line 4.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/john/samples/hello

Breakpoint 1, main () at hello.c:4
4               printf("Hello world!\n");
(gdb) monitor target

 LEON SPARC simulator backend, version 1.0 (professional version)

 Copyright (C) 2001, Gaisler Research - all rights reserved.
using 64-bit time
serial port A on stdin/stdout
allocated 4096 K RAM memory, in 1 bank(s)
allocated 2048 K ROM memory
icache: 1 * 4 kbytes, 16 bytes/line (4 kbytes total)
dcache: 1 * 4 kbytes, 16 bytes/line (4 kbytes total)

 LEON SPARC simulator backend, version 1.0 (professional version)

 Copyright (C) 2001, Gaisler Research - all rights reserved.
using 64-bit time
(gdb) lo
Loading section .text, size 0x3940 lma 0x40000000
Loading section .data, size 0x750 lma 0x40003940
Start address 0x40000000, load size 16528
Transfer rate: 132224 bits in <1 sec, 275 bytes/write.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/john/samples/hello
```

```
Breakpoint 1, main () at hello.c:4
4               printf("Hello world!\n");
(gdb) cont
Continuing.

Program exited normally.
```

4.2 Debugging of applications

To load and start an application, use the gdb **load** and **run** command.

```
(gdb) lo
Loading section .text, size 0xcb90 lma 0x40000000
Loading section .data, size 0x770 lma 0x4000cb90
Start address 0x40000000, load size 54016
Transfer rate: 61732 bits/sec, 278 bytes/write.
(gdb) bre main
Breakpoint 1 at 0x400039c4: file stanford.c, line 1033.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/john/samples/stanford

Breakpoint 1, main () at stanford.c:1033
1033        fixed = 0.0;
(gdb)
```

To interrupt simulation, Ctrl-C can be typed in both GDB and GRMON windows. The program can be restarted using the GDB **run** command but a **load** has first to be executed to reload the program image on the target. Software trap 1 (ta 1) is used by gdb to insert breakpoints and should not be used by the application.

4.3 Detaching

If gdb is detached using the **detach** command, the monitor returns to the command prompt, and the program can be debugged using the standard GRMON commands. The monitor can also be re-attached to gdb by issuing the **gdb** command to the monitor (and the **target** command to gdb).

GRMON translates SPARC traps into (unix) signals which are properly communicated to gdb. If the application encounters a fatal trap, execution will be stopped exactly before the failing instruction. The target memory and register values can then be examined in gdb to determine the error cause.

4.4 Limitations of gdb interface

For optimal operation, gdb-5.3 configured for grmon should be used (provided with leccs-1.1.5.3 or later).

Do not use the gdb **where** command in parts of an application where traps are disabled (e.g.trap handlers). Since the stack pointer is not valid at this point, gdb might go into an infinite loop trying to unwind false stack frames.

5 The DSU backend

5.1 General

The DSU backend is a debug monitor for the LEON processor debug support unit. It includes the following functions:

- Read/write access to all LEON registers and memory
- Built-in disassembler and trace buffer management
- Downloading and execution of LEON applications
- Breakpoint and watchpoint management
- Remote connection to GNU debugger (gdb)
- Auto-probing and initialisation of LEON peripherals and memory settings

When referring to GRMON in this section, it implies the simulator backend only.

5.2 Operation

5.2.1 Overview

The LEON DSU can be controlled through any AHB master, and the DSU backend supports communications through the dedicated DSU uart or (if available) a PCI interface.

5.2.2 Starting GRMON/DSU using the DSU uart

To succefully attach GRMON with the DSU backend using a uart, first connect the serial cable between the target board and the host system, then power-up and reset the target board, and finally start GRMON. Use the -uart option in case the DSU is not connected to /dev/ttyS0 of your host. Note that the DSUEN signal on the LEON processor has to be asserted for the DSU to operate.

When the DSU backend first connects to the target, a check is made to see if the system has been initialised with respect to memory, UART and timer settings. If no initialisation has been made (= debug mode entered directly after reset), the system first has to be initialised before any application can run. This is performed automatically by probing for available memory banks, and detecting the system frequency. The initialisation can also be forced by giving the -i switch at startup. The de
tected system settings are printed on the console:

```
john@pluto:tmp/grmon% grmon -i

 GRMON - The LEON multi purpose monitor v1.0

 Copyright (C) 2004, Gaisler Research - all rights reserved.
 For latest updates, go to http://www.gaisler.com/
 Comments or bug-reports to grmon@gaisler.com

 LEON DSU Monitor backend 1.0  (professional version)

 Copyright (C) 2003, Gaisler Research - all rights reserved.
 Comments or bug-reports to jiri@gaisler.com

 using port /dev/ttyS0 @ 115200 baud
```

```
processor frequency   :  99.53 MHz
register windows      :  8
v8 hardware mul/div   :  yes
floating-point unit   :  meiko
instruction cache     :  1 * 8 kbytes, 32 bytes/line (8 kbytes total)
data cache            :  1 * 8 kbytes, 32 bytes/line (8 kbytes total)
hardware breakpoints  :  4
trace buffer          :  256 lines, mixed cpu/ahb tracing
sram width            :  32 bits
sram banks            :  1
sram bank size        :  1024 kbytes
sdram                 :  1 * 64 Mbyte @ 0x60000000
sdram parameters      :  column bits: 9, cas delay: 2, refresh 15.6 us
stack pointer         :  0x400ffff0
```

After monitor initialisation, the current value of MCFG1-3 and stack pointer are assigned as the default values, to which the registers and stack pointer will reset before a new application is started. These default valuse can however be changed with the **mcfg1/2/3** and **stack** commands.

5.2.3 Starting GRMON/DSU using a PCI interface (Linux only)

If your LEON processor has a PCI target interface, GRMON can connect to the LEON DSU using the PCI bus. In this case, start GRMON with -pci or -pcidev # (see options below). The PCI interfaces uses the open-source PHOB generic device driver for linux, which must loaded before GRMON is started:

```
root@mars:~/phob-1.0# ./phob_load vendor_id=0x16e3 device_id=0x0210
```

When the PHOB driver is loaded, make sure that the corresponding devices are writable by the user. The driver includes a script (phob_load) that can be edited for the correct chmod operation.

Once the driver is loaded, start GRMON with the -pci switch:

```
dsu port is /dev/phob0 (PCI)

clock frequency     : 24.73 MHz
register windows    : 8
instruction cache   : 4 * 2 kbytes, 32 bytes/line (8 kbytes total)
data cache          : 4 * 2 kbytes, 32 bytes/line (8 kbytes total)
trace buffer        : 256 lines, mixed cpu/ahb tracing
PCI core            : opencores (16e3:0210)
sdram               : 1 * 32 Mbyte @ 0x40000000
sdram parameters    : column bits: 9, cas delay: 2, refresh 15.5 us
stack pointer       : 0x41fffff0
UART 1 in DSU mode
grmon[dsu]>
```

5.2.4 Command line options

There are a few command line options to grmon that is specific for the DSU backend. These are:

-abaud *baudrate*

Use *baudrate* for UART 1 & 2. By default, 38400 baud is used.

-**banks** *ram_banks*

Overrides the auto-probed number of populated ram banks.

-baud *baudrate*

Use *baudrate* for the DSU serial link. By default, 115200 baud is used.

-**cas** *delay* Programs SDRAM to either 2 or 3 cycles CAS delay. Default is 2.

-**da** *addr* DSU address. By default, dsumon expects that the LEON DSU is located at AHB address 0x90000000. Use this option if you have configured the DSU to use a different address.

-**freq** *system_clock*

Overrides the detected system frequency. Use with care!

-ibaud *baudrate*

Use *baudrate* to determine the target processor frequency. Lower rate means higher accuracy. The detected frequency is printed on the console during startup. By default, 115200 baud is used.

-**nb** Do not break on error traps, i.e. set the BZ bit to 0 in the DSU control register. This setting is necessary if runing linux or other programs that use the data exception trap (tt=0x09).

-**nosram** Disable sram and map sdram from address 0x40000000

-**ram** *ram_size*

Overrides the auto-probed amount of static ram. Size is given in Kbytes.

-**romrws** *waitstates*

Set *waitstates* number of waitstates for rom reads.

-**romwws** *waitstates*

Set *waitstates* number of waitstates for rom writes.

-**romws** *waitstates*

Set *waitstates* number of waitstates for both rom reads and writes.

-**ramrws** *waitstates*

Set *waitstates* number of waitstates for ram reads.

-**ramwws** *waitstates*

Set *waitstates* number of waitstates for ram writes.

-**ramws** *waitstates*

Set *waitstates* number of waitstates for both ram reads and writes.

-**pci** Connect to the DSU using PCI device /dev/phob0.
(See -uart option for description of how to connect to different devices)

-stack *stackval*

Set *stackval* as stack pointer for applications, overriding the auto-detected value.

-**uart** *device* By default, DSUMON communicates with the target using /dev/ttyS0 (/dev/ttya on solaris). This switch can be used to connect to the target using other devices. e.g '-uart /dev/cua0. Note that -uart is also used when more than one board is handled by the phob driver, e.g.:
grmon -i -pci -uart /dev/phob0afor 1st board
grmon -i -pci -uart /dev/phob1afor 2nd board

-**u** Put UART 1 in loop-back mode, and print its output on monitor console.

5.3 Interactive mode

5.3.1 Commands specific to the DSU backend

These are the commands only available in the DSU backend:

ahb [*length*] Print only the AHB trace buffer. The *length* last AHB transfers will be printed, default is 10.

hist [*length*] Print the trace buffer. The *length* last executed instructions or AHB transfers will be printed, default is 10.

inst [*length*] Print only the instruction trace buffer. The *length* last executed instructions will be printed, default is 10.

init Do a hardware probe to detect system settings and memory size, and initialize peripherals.

leon Display LEON peripherals registers.

mcfg1 [*value*] Set the default value for memory configuration register 1. When the 'run' command is given, MCFG1, 2&3 are initialised with their default values to provide the application with a clean startup environment. If no value is give, the current value is printed.

mcfg2 [*value*] As mcfg1 above, but setting the default value of the MCFG2 register.

mcfg3 [*value*] As mcfg1 above, but setting the default value of the MCFG3 register.

mmu Prints the MMU registers

reg [*reg_name value*]

Prints and sets the IU registers in the current register window. **reg** without parameters prints the IU registers. **reg** *reg_name value* sets the corresponding register to *value*. Valid register names are psr, tbr, wim, y, g1-g7, o0-o7 and l0-l7. To view the other register windows, use **reg** *w*n, where n is 0 - 7.

stack [*value*] Set the default value of the stack pointer. The given value will be aligned to nearest lower

256-byte boundary. If no value is given, the current value is printed.

tmode [*proc* | *ahb* | *both* | *none*]

Select tracing mode between none, processor-only, AHB only or both.

va <*address*> Performs a virtual-to-physical translation of *address*.

When used with Gaisler Research's GR-PCI-XC2V LEON FPGA Development board, the following commands are also supported. Note that flash has to be enabled with 'flash enable' before any other flash commands can be used.

flash Print the on-board flash memory configuration

flash disable Disable writing to flash

flash enable Enable writing to flash

flash erase [addr | all]

Erase a flash block at address **addr**, or the complete flash memory (**all**). An address range is also support,e.g. 'flash erase 0x1000 0x8000'.

flash lock [addr | all]

Lock a flash block at address **addr**, or the complete flash memory (**all**). An address range is also support, e.g. 'flash erase 0x1000 0x8000'.

flash lockdown [addr | all]

Lock-down a flash block at address **addr**, or the complete flash memory (**all**). An address range is also support, e.g. 'flash erase 0x1000 0x8000'.

flash query Print the flash query registers

flash status Print the flash lock status register

flash unlock [addr | all]

Unock a flash block at address **addr**, or the complete flash memory (**all**). An address range is also support, e.g. 'flash erase 0x1000 0x8000'.

flash write <*addr*> <*data*>

Write a 32-bit data word to the flash at address **addr**.

5.3.2 Using the trace buffer

Depending on the LEON configuration, the trace buffer can store the last executed instruction, the last AHB bus transfers, or both. The trace buffer mode is set using the **tmode** command. Use the ahb, inst or hist commands to display the contents of the buffer. Below is an example debug session that shows the usage of breakpoints, watchpoints and the trace buffer:

```
john@pluto:tmp/grmon-0.1% grmon -i

 GRMON - The LEON multi purpose monitor v1.0

 Copyright (C) 2004, Gaisler Research - all rights reserved.
 For latest updates, go to http://www.gaisler.com/
 Comments or bug-reports to grmon@gaisler.com

 LEON DSU Monitor backend 1.0  (professional version)

 Copyright (C) 2003, Gaisler Research - all rights reserved.
 Comments or bug-reports to jiri@gaisler.com

 using port /dev/ttyS0 @ 115200 baud
```

```
 processor frequency   :  99.53 MHz
 register windows      :  8
 v8 hardware mul/div   :  yes
 floating-point unit   :  meiko
 instruction cache     :  1 * 8 kbytes, 32 bytes/line (8 kbytes total)
 data cache            :  1 * 8 kbytes, 32 bytes/line (8 kbytes total)
 hardware breakpoints  :  4
 trace buffer          :  256 lines, mixed cpu/ahb tracing
 sram width            :  32 bits
 sram banks            :  1
 sram bank size        :  1024 kbytes
 sdram                 :  1 * 64 Mbyte @ 0x60000000
 sdram parameters      :  column bits: 9, cas delay: 2, refresh 15.6 us
 stack pointer         :  0x400ffff0
grmon[dsu]> lo samples/stanford
section: .text at 0x40000000, size 41168 bytes
section: .data at 0x4000a0d0, size 1904 bytes
total size: 43072 bytes (94.2 kbit/s)
read 158 symbols
grmon[dsu]> tm both
combined processor/AHB tracing
grmon[dsu]> break F
Fft     Fit     fflush  free    fstat
grmon[dsu]> break Fft
grmon[dsu]> watch  0x4000a500
grmon[dsu]> bre
num   address        type
  1 : 0x40003608    (soft)
  2 : 0x4000a500    (watch)
grmon[dsu]> run
watchpoint 2  free + 0x1c8 (0x400042d0)
grmon[dsu]> ah
   time      address   type   data     trans size burst mst lock resp  tt  pil irl
  239371457  400042c4  read   38800002   3    2    1    0    0    0   06   0   0
  239371459  400042c8  read   d222a0fc   3    2    1    0    0    0   06   0   0
  239371461  400042cc  read   15100029   3    2    1    0    0    0   06   0   0
  239371463  400042d0  read   d002a100   3    2    1    0    0    0   06   0   0
  239371465  400042d4  read   80a24008   3    2    1    0    0    0   06   0   0
  239371467  400042d8  read   38800002   3    2    1    0    0    0   06   0   0
  239371469  400042dc  read   d222a100   3    2    1    0    0    0   06   0   0
  239371472  4000a4fc  read   00000000   2    2    0    0    0    0   06   0   0
  239371480  4000a4fc  write  000005d0   2    2    0    0    0    0   06   0   0
  239371481  90000000  read   000055f9   2    2    0    3    0    0   06   0   0
grmon[dsu]> in
   time      address   instruction                result
  239371441  40004254  cmp  %l3, %o2               [00000000]
  239371446  40004258  be  0x400042b4              [00000000]
  239371451  4000425c  st  %o0, [%l1 + 0x4]        [40021a34 000005d1]
  239371456  400042b4  sethi  %hi(0x4000a400), %o2 [4000a400]
  239371457  400042b8  ld  [%l6 + 0x104], %o1      [000005d0]
  239371473  400042bc  ld  [%o2 + 0xfc], %o0       [00000000]
  239371475  400042c0  cmp  %o1, %o0               [000005d0]
  239371476  400042c4  bgu,a  0x400042cc           [00000000]
  239371478  400042c8  st  %o1, [%o2 + 0xfc]       [4000a4fc 000005d0]
  239371479  400042cc  sethi  %hi(0x4000a400), %o2 [4000a400]
grmon[dsu]> del 2
grmon[dsu]> break
num   address        type
  1 : 0x40003608    (soft)
grmon[dsu]> cont
breakpoint 1  Fft (0x40003608)
```

```
grmon[dsu]> hi
  254992752  4000386c  sethi  %hi(0x40014400), %l1 [40014400]
  254992753            ahb read,  mst=0, size=2    [4000387c 9214a28c]
  254992755  40003870  sethi  %hi(0x4001f800), %l0 [4001f800]
  254992759            ahb read,  mst=0, size=2    [40003880 94146198]
  254992760  40003874  mov  19, %i0                [00000013]
  254992761            ahb read,  mst=0, size=2    [40003884 961423cc]
  254992762  40003878  mov  256, %o0               [00000100]
  254992763            ahb read,  mst=0, size=2    [40003888 190fec00]
  254992764  4000387c  or  %l2, 0x28c, %o1         [40014e8c]
  254992765            ahb read,  mst=0, size=2    [4000388c 7fffff5f]
  254992766  40003880  or  %l1, 0x198, %o2         [40014598]
  254992767            ahb read,  mst=0, size=2    [40003890 9a102000]
  254992769            ahb read,  mst=0, size=2    [40003894 b0863fff]
  254992771  40003884  or  %l0, 0x3cc, %o3         [4001fbcc]
  254992772  40003888  sethi  %hi(0x3fb00000), %o4 [3fb00000]
  254992773  4000388c  call  0x40003608            [4000388c]
  254992774  40003890  mov  0, %o5                 [00000000]
grmon[dsu]> delete 1
grmon[dsu]> cont

Program exited normally.
grmon[dsu]>
```

When printing executed instructions, the value within brackets denotes the instruction result, or in the case of store instructions the store address and store data. The value in the first column displays the relative time, equal to the DSU timer. The time is taken when the instruction completes in the last pipeline stage (write-back) of the processor. In a mixed instruction/AHB display, AHB address and read or write value appear within brackets. The time indicates when the transfer completed, i.e. when HREADY was asserted. Note:, when switching between tracing modes the contents of the trace buffer will not be valid until execution has been resumed and the buffer refilled.

5.3.3 Forwarding application console output

If GRMON is started with **-u**, the LEON UART1 will be placed in loop-back mode, with flow-control enabled. During the execution of an application, the UART receiver will be regularly polled, and all application console output will be printed on the GRMON console. It is then not necessary to connect a separate terminal to UART1 to see the application output. NOTE: the applications must be compiled with LECCS-1.1.5 or later, and LEON processor 1.0.4 or later must be used for this function to work.

5.4 MMU support

If the LEON MMU is configured and enabled, GRMON performs automatic virtual to physical address translation when displaying disassembly and memory contents. Writing to memory using 'wmem' is always done using the physical address.

5.5 GDB interface

5.5.1 Some gdb support functions

GRMON detects gdb access to register window frames in memory which are not yet flushed and only reside in the processor register file. When such a memory location is read, DSUMON will read the correct value from the register file instead of the memory. This allows gdb to form a function traceback without any (intrusive)

modification of memory. This feature is disabled during debugging of code where traps are disabled, since not valid stack frame exist at that point.

GRMON detects the insertion of gdb breakpoints, in form of the '*ta 1*' instruction. When a breakpoint is inserted, the corresponding instruction cache tag is examined, and if the memory location was cached the tag is cleared to keep memory and cache synchronised.

5.5.2 MMU support

If the LEON MMU is configured and enabled, GRMON automatically performs virtual to physical address translation when responding to gdb requests. This allows souce-level debugging of programs using the MMU, such as linux. The gdb set memory command is however not supported.

6 The simulator backend

6.1 General

The GRMON simulator backend is a generic SPARC[1] architecture simulator capable of emulating LEON-based computer systems.

The simulator backend provides several unique features:

- Accurate and cycle-true emulation of LEON processors
- Accelerated simulation during processor standby mode
- 64-bit time for unlimited simulation periods
- Instruction trace buffer
- Loadable modules to include user-defined I/O devices
- Stack backtrace with symbolic information
- Check-pointing capability to save and restore complete simulator state

When referring to GRMON in this section, it implies the simulator backend only.

6.2 Operation

6.2.1 Command line options

The following command line options are unique for the simulator backend:

-**ahbm** *ahb_module_path*

Use *ahb_module*_path as loadable AHB module rather than the default ahb.so

-**banks** *ram_banks*

Sets how many ram banks (1 - 4) the ram is divided on. Default is 1.

-**bopt** Enables idle-loop optimisation (see text).

-**cpm** *cp_module*

Use *cp_module* as loadable co-processor module rather than the default cp.so.

-**dcsize** *size* Defines the set-size (kbytes) of the LEON dcache. Allowed values are 1 - 64 in binary steps.

-**dlock** Enable data cache line locking. Default if disabled.

-**dlsize** *size* Sets the line size of the LEON data cache (in bytes). Allowed values are 8, 16 or 32.

-**dsets** *sets* Defines the number of sets in the LEON data cache. Allowed values are 1 (default) - 4.

-**drepl** *repl* Sets the replacement algorithm for the LEON data cache. Allowed values are rnd (default) for random replacement, lru for the least-recently-used replacement algorithm and lru for Least-recently replacement algorithm.

1. SPARC is a registred trademark of SPARC International

-**freq** *system_clock*

Sets the simulated system clock (MHz). Will affect UART timing and performance statistics. Default is 50.

-**fast_uart** Run UARTS at infinite speed, rather than with correct (slow) baud rate.

-**fpm** *fp_module*

Use *fp_module* as loadable FPU module rather than the default fp.so.

-**icsize** *size* Defines the set-size (kbytes) of the LEON icache. Allowed values are 1 - 64 in binary steps.

-**isets** *sets* Defines the number of sets in the LEON instruction cache. Allowed values are 1(default) - 4.

-**ilock** Enable instruction cache line locking. Default is disabled.

-**iom** *io_module*

Use *io_module* as loadable I/O module rather than the default io.so.

-**ilsize** *size* Sets the line size of the LEON instruction cache (in bytes). Allowed values are 8, 16 or 32.

-**irepl** *repl* Sets the replacement algorithm for the LEON instruction cache. Allowed values are rnd (default) for random replacement, lru for the least-recently-used replacement algorithm and lrr for least-recently- replacement algorithm.

-**nfp** Disables the FPU to emulate system without FP hardware. Any FP instruction will generate an FPdisabled trap.

-**nomac** Disable LEON MAC instruction.

-**nov8** Disable SPARC V8 MUL/DIV instructions.

-**notimers** Disable the LEON timer unit.

-**nouart** Disable emulation of UARTS. All access to UART registers will be routed to the I/O module.

-**ram** *ram_size*

Sets the amount of simulated RAM (Kbyte). Default is 4096. If set to 0, sram is disabled (disable available on LEON only).

-sdram *sdram_size*

Sets the amount of simulated SDRAM. Default is 0. To use SDRAM, relevant memory configuration registers must be initialised. (LEON only)

-**rom** *rom_size*

Sets the amount of simulated ROM (Kbyte). Default is 2048.

-**rom8, -rom16**

By default, the prom area at reset time is considered to be 32-bit. Specifying -rom8 or -rom16 will initialise the memory width field in the memory configuration register to 8- or 16-bits. The only visible difference is in the instruction timing.

-**uart[1,2]** *device*

By default, UART1 is connected to stdin/stdout and UART2 is disconnected. This switch can be used to connect the uarts to other devices. E.g., '-uart1 /dev/ptypc' will attach UART1 to ptypc.

6.2.2 Commands specific for the simulator backend

These are the commands only available in the simulator backend:

aprof [*0|1*] [*period*]

Enable/disable accumulative profiling. (see "Profiling" on page 29)

bt Print backtrace

cp Shows the registers of the co-processor (if available).

event Print events in the event queue. Only user-inserted events are printed.

flush [*all* | *icache* | *dcache* | *addr*]

Flush the LEON caches. Specifying *all* will flush both the icache and dcache. Specifying *icache* or *dcache* will flush the respective cache. Specifying *addr* will flush the corresponding line in both caches.

hist [*length*]Enable the instruction trace buffer. The *length* last executed instructions will be placed in the trace buffer. A **hist** command without *length* will display the trace buffer. Specifying a zero trace length will disable the trace buffer.

inc *time* Increment simulator time without executing instructions. Time is given in same format as for the **go** command. Event queue is evaluated during the advancement of time.

perf [*reset*] The **perf** command will display various execution statistics. A 'perf reset' command will reset the statistics. This can be used if statistics shall be calculated only over a part of the program. The **run** and **reset** command also resets the statistic information.

restore *file* Restores simulator state from *file*. (see "Check-pointing" on page 29)

save *file* Saves simulator state to *file*. (see "Check-pointing" on page 29)

6.2.3 Backtrace

The bt command will display the current call backtrace and associated stack pointer;

```
grmon[sim]> load samples/hello
section: .text at 0x40000000, size 14656 bytes
section: .data at 0x40003940, size 1872 bytes
total size: 16528 bytes (in <1 sec)
read 71 symbols
grmon[sim]> break _puts_r
breakpoint 1 at 0x40001dcc:  _puts_r
grmon[sim]> run
resuming at 0x40000000
breakpoint 1  _puts_r (0x40001dcc)
grmon[sim]> bt
    %pc          %sp
#0   0x40001dcc   0x403ffd10   _puts_r + 0x0
#1   0x40001e3c   0x403ffd98   puts + 0x8
#2   0x40001208   0x403ffe00   _start + 0x60
```

```
#3   0x40001014   0x403ffe40   start + 0x1014
grmon[sim]>
```

6.2.4 Check-pointing

The **professional** version of the simulator backend can save and restore its complete state, allowing to resume simulation from a saved check-point. Saving the state is done with the **save** command:

grmon[sim]> save *file_name*

The state is save to *file_name*.tss. To restore the state, use the **restore** command:

grmon[sim]> restore *file_name*

The state will be restored from *file_name*.tss . Restore directly at startup can be performed with the '**-rest** *file_name'* command line switch.

Note that GRMON command line options are not stored (such as alternate UART devices, etc.).

6.2.5 Profiling

The profiling function calculates the amount of execution time spent in each subroutine of the simulated program. This is made without intervention or instrumentation of the code by periodically sample the execution point and the associated call tree. Cycles in the call graph are properly handled, as well as sections of the code where no stack is available (e.g. trap handlers). The profiling information is printed as a list sorted on highest execution time ration. Profiling is enabled through the aprof command. The sampling period is by default 1000 clocks which typically provides the best compromise between accuracy and performance. Other sampling periods can also be set through the aprof command. Below is an example profiling the stanford benchmark:

```
> ./grmon -sim samples/stanford

 GRMON - The LEON multi purpose monitor v1.0.5

 Copyright (C) 2004, Gaisler Research - all rights reserved.
 For latest updates, go to http://www.gaisler.com/
 Comments or bug-reports to grmon@gaisler.com


 LEON SPARC simulator backend, version 1.0.5

 Copyright (C) 2001, Gaisler Research - all rights reserved.
serial port A on stdin/stdout
allocated 4096 K RAM memory, in 1 bank(s)
allocated 2048 K ROM memory
icache: 1 * 4 kbytes, 16 bytes/line (4 kbytes total)
dcache: 1 * 4 kbytes, 16 bytes/line (4 kbytes total)
section: .text at 0x40000000, size 52112 bytes
section: .data at 0x4000cb90, size 1904 bytes
total size: 54016 bytes (in <1 sec)
read 195 symbols
grmon[sim]> aprof 1
Accumulated profiling enabled, sample period 1000
grmon[sim]> run
resuming at 0x40000000
Starting
    Perm  Towers  Queens   Intmm      Mm  Puzzle   Quick  Bubble    Tree     FFT
      50      33      17     116    1100     217      33      34     266     934
```

```
Nonfloating point composite is        126

Floating point composite is        862

Program exited normally.
grmon[sim]> aprof
function                  samples     ratio(%)
start                      135242      48.44
_start                     131006      46.92
main                       130790      46.84
__mulsf3                    77355      27.70
rInnerproduct               54288      19.44
Mm                          54200      19.41
Fft                         47447      16.99
__unpack_f                  47254      16.92
Oscar                       46411      16.62
__pack_f                    34102      12.21
__addsf3                    29967      10.73
__divdi3                    28324      10.14
__muldi3                    18526       6.63
.umul                       17824       6.38
__subsf3                    15547       5.56
Fit                         15188       5.44
Trial                       11523       4.12
Puzzle                      11004       3.94
--- a lot more output ---
grmon[sim]>
```

6.3 Emulation characteristics

6.3.1 Timing

The simulator backend is cycle-true, i.e a simulator time is maintained and incremented according processor instruction timing and memory latency. Tracing using the **trace** command will display the current simulator time in the left column. This time indicates when the instruction is fetched. Cache misses, waitstates or data dependencies will delay the following fetch according to the incurred delay.

6.3.2 UARTS

If the baudrate register is written by the application software, the UARTS will operate with correct timing. If the baudrate is left at the default value, or if the **-fast_uart** switch was used, the UARTS operate at infinite speed. This means that the transmitter holding register always is empty and a transmitter empty interrupt is generated directly after each write to the transmitter data register. The receivers can never overflow or generate errors.

Note that with correct UART timing, it is possible that the last character of a program is not displayed on the console. This can happen if the program forces the processor in error mode, thereby terminating the simulation, before the last character has been shifted out from the transmitter shift register. To avoid this, an application should poll the UART status register and not force the processor in error mode before the transmitter shift registers are empty. The real hardware does not exhibit this problem since the UARTs continue to operate even when the processor is halted.

6.3.3 FPU

The simulator maps floating-point operations on the hosts floating point capabilities. This means that accuracy and generation of IEEE exceptions is host dependent. The simulator implements (to some extent) data-dependant execution timing as in the real MEKIO FPU.

6.3.4 Delayed write to special registers

The SPARC architecture defines that a write to the special registers (%psr, %wim, %tbr, %fsr, %y) may have up to 3 delay cycles, meaning that up to 3 of the instructions following a special register write might not 'see' the newly written value due to pipeline effects. While LEON have between 2 and 3 delay cycles, the GRMON simulator backend has 0. This does not affect simulation accuracy or timing as long as the SPARC ABI recommendations are followed that each special register write must always be followed by three NOP. If the three NOP are left out, the software might fail on real hardware while still executing 'correctly' on the simulator.

6.3.5 Idle-loop optimisation

To minimise power consumption, LEON applications will typically place the processor in power-down mode when the idle task is scheduled in the operation system. In power-down mode, GRMON increments the event queue without executing any instructions, thereby significantly improving simulation performance. However, some (poorly written) code might use a busy loop (BA 0) instead of triggering power-down mode. The **-bopt** switch will enable a detection mechanism which will identify such behavior and optimise the simulation as if the power-down mode was entered.

6.3.6 Processor timing

The GRMON simulator backend emulates the behavior of the LEON-2.2 VHDL model.

6.3.7 Cache memories

The evaluation version of LEON implements 2*4Kbyte caches, with 16 bytes per line. The commercial GRMON simulator backend version can emulate any permissible cache configuration using the -icsize, -ilsize, -dcsize and -dlsize options. Allowed sizes are 1 - 64 kbyte with 8 - 32 bytes/line. The characteristics leon multi-set caches (as of leon2-1.0.8) can be emulated using the -isets, -dsets, -irepl, -drelp, -ilock and -dlock options. Diagnostic cache reads/writes are implemented. The simulator commands **icache** and **dcache** can be used to display cache contents.

6.3.8 LEON peripherals registers

The LEON peripherals registers can be displayed with the **leon** command, or using **x** ('x 0x80000000 256'). The registers can also be written using **wmem** (e.g. 'wmem 0x80000000 0x1234').

6.3.9 Interrupt controller

External interrupts are not implemented, so the I/O port interrupt register has no function. Internal interrupts are generated as defined in the LEON specification. All 15 interrupts can also be generated from the user-defined I/O module using the set_irq() callback.

6.3.10 Power-down mode

The power-down register 0x80000018) is implemented as in the specification. A Ctrl-C in the simulator window will exit the power-down mode. In power-down mode, the simulator skips time until the next event in the event queue, thereby significantly increasing the simulation speed.

6.3.11 Memory emulation

The memory configuration registers 1/2 are used to decode the simulated memory. The memory configuration registers has to be programmed by software to reflect the available memory, and the number and size of the memory banks. This waitstates fields must also be programmed with the correct configuration after reset.

Using the **-banks** option, it is possible to set over how many ram banks the external ram is divided in. Note that software compiled with LECCS, and **not** run through mkprom must **not** use this option. For mkprom encapsulated programs, it is essential that the **same** ram size and bank number setting is used for both mkprom and GRMON.

6.3.12 SPARC V8 MUL/DIV/MAC instructions

GRMON supports the SPARC V8 multiply, divide and MAC instruction. To correctly emulate LEON processors which do not implement these instructions, use the **-nomac** to disable the MAC instruction or -**nov8** to disable multiply and divide instructions.

6.4 Loadable modules

6.4.1 The simulator backend I/O emulation interface

User-defined I/O devices can be loaded into the simulator through the use of loadable modules. As the real processor, the simulator primarily interacts with the emulated device through read and write requests, while the emulated device can optionally generate interrupts and DMA requests. This is implemented through module interface described below. The interface is made up of two parts; one that is exported by GRMON and defines simulator functions and data structures that can be used by the I/O device; and one that is exported by the I/O device and allows GRMON to access the I/O device. Address decoding of the I/O devices is straight-forward: all access that do not map on the internally emulated memory and control registers are forwarded to the I/O module.

The simulator backend exports two structures: simif and ioif. The simif structure defines functions and data structures belonging to the simulator core, while ioif defines functions provided by system (LEON) emulation. At start-up, if GRMON is started with the in simulator mode, it searches for 'io.so' in the current directory, but the location of the module can be specified using the **-iom** switch. Note that the module must be compiled to be position-independent, i.e. with the -fPIC switch (gcc).

6.4.1.1 simif structure

The simif structure is defined in sim.h:

```
struct sim_options {
    int phys_ram;
    int phys_rom;
    double freq;
    double wdfreq;
};
struct sim_interface {
    struct sim_options *options; /* tsim command-line options */
    sistime *simtime;     /* current simulator time */
    void (*event)(void (*cfunc)(), int arg, sistime offset);
```

```
    void (*stop_event)(void (*cfunc)());
    int *irl;       /* interrup request level */
    void (*sys_reset)();  /* reset processor */
    void (*sim_stop)();   /* stop simulation */ };
    extern struct sim_interface simif;/* exported simulator functions */
}
```

The elements in the structure has the following meaning:

```
struct sim_options *options;
```

Contains some simulator startup options. options.freq defines the clock frequency of the emulated processor and can be used to correlate the simulator time to the real time.

```
sistime *simtime;
```

Contains the current simulator time. Time is counted in clock cycles since start of simulation. To calculate the elapsed real time, divide simtime with options.freq.

```
void (*event)(void (*cfunc)(), int arg, sistime offset);
```

GRMON maintains an event queue to emulate time-dependant functions. The event() function inserts an event in the event queue. An event consists of a function to be called when the event expires, an argument with which the function is called, and an offset (relative the current time) defining when the event should expire. NOTE: the event() function may NOT be called from a signal handler installed by the I/O module, but only from event() callbacks or at start of simulation.

```
void (*stop_event)(void (*cfunc)());
```

stop_event() will remove all events from the event queue which has the calling function equal to cfunc(). NOTE: the
stop_event() function may NOT be called from a signal handler installed by the I/O module.

```
int *irl;
```

Current IU interrupt level. Should not be used by I/O functions unless they explicitly monitor theses lines.

```
void (*sys_reset)();
```

Performs a system reset. Should only be used if the I/O device is capable of driving the reset input.

```
void (*sim_stop)();
```

Stops current simulation. Can be used for debugging purposes if manual intervention is needed after a certain event.

6.4.1.2 ioif structure

ioif is defined in sim.h:

```
structio_interface {
    void (*set_irq)(int irq, int level);
```

```
    int (*dma_read)(uint32 addr, uint32 *data, int num);
    int (*dma_write)(uint32 addr, uint32 *data, int num);
};

extern struct io_interface ioif; /* exported processor interface */
```

The elements of the structure have the following meaning:

```
void (*set_irq)(int irq, int level);
```

Set the interrupt pending bit for interrupt irq. Valid values on irq is 1 - 15. Care should be taken not to set interrupts used by the LEON emulated peripherals. Note that the LEON interrupt control register controls how and when processor interrupts are actually generated.

```
int (*dma_read)(uint32 addr, uint32 *data, int num);
int (*dma_write)(uint32 addr, uint32 *data, int num);
```

Performs DMA transactions to/from the emulated processor memory. Only 32-bit word transfers are allowed, and the address must be word aligned. On bus error, 1 is returned, otherwise 0. The DMA takes place on the AMBA AHB bus.

6.4.1.3 Structure to be provided by I/O device

io.h defines the structure to be provided by the emulated I/O device:

```
struct io_subsystem {
    void (*io_init)();/* called once on start-up */
    void (*io_exit)();/* called once on exit */
    void (*io_reset)();/* called on processor reset */
    void (*io_restart)();/* called on simulator restart */
    int (*io_read)(unsigned int addr, int *data, int *ws);
    int (*io_write)(unsigned int addr, int *data, int *ws, int size);
    char *(*get_io_ptr)(unsigned int addr, int size);
    void (*command)(char * cmd);/* I/O specific commands */
    void (*sigio)();/* called when SIGIO occurs */
    void (*save)(char *fname);/* save simulation state */
    void (*restore)(char *fname);/* restore simulation state */
};
extern struct io_subsystem *io; /* imported I/O emulation functions */
```

The elements of the structure have the following meanings:

```
void (*io_init)();
```

Called once on simulator startup. Set to NULL if unused.

```
void (*io_exit)();
```

Called once on simulator exit. Set to NULL if unused.

```
void (*io_reset)();
```

Called every time the processor is reset (i.e also startup). Set to NULL if unused.

```
void (*io_restart)();
```

Called every time the simulator is restarted (simtime set to zero). Set to NULL if unused.

```
int (*io_read)(unsigned int addr, int *data, int *ws);
```

Processor read call. The processor always reads one full 32-bit word from addr. The data should be returned in *data, the number of waitstates should be returned in *ws. If the access would fail (illegal address etc.), 1 should be returned, on success 0.

```
int (*io_write)(unsigned int addr, int *data, int *ws, int size);
```

Processor write call. The size of the written data is indicated in size: 0 = byte, 1 = half-word, 2 = word, 3 = doubleword. The address is provided in addr, and is always aligned with respect to the size of the written data. The number of waitstates should be returned in *ws. If the access would fail (illegal address etc.), 1 should be returned, on success 0.

```
char * (*get_io_ptr)(unsigned int addr, int size);
```

GRMON can access emulated memory in the I/O device in two ways: either through the io_read/io_write functions or directly through a memory pointer. get_io_ptr() is called with the target address and transfer size, and should return a character pointer to the emulated memory array if the address and size is within the range of the emulated memory. If outside the range, -1 should be returned. Set to NULL if not used.

```
int (*command)(char * cmd);
```

The I/O module can optionally receive command-line commands. A command is first sent to the AHB and I/O modules, and if not recognised, the to GRMON. command() is called with the full command string in *cmd. Should return 1 if the command is recognized, otherwise 0. When grmon commands are issued through the gdb 'monitor' command, a return value of 0 or 1 will result in an 'OK' response to the gdb command. A return value > 1 will send the value itself as the gdb response. A return value < 1 will truncate the lsb 8 bits and send them back as a gdb error response: 'E*nn*'.

```
void (*save)(char *fname);/* save simulation state */
```

The save() function is called when save command is issued in the simulator. The I/O module should save any required state which is needed to completely restore the state at a later stage. *fname points to the base file name which is used by GRMON - GRMON save the internal state of the simulator backend to fname.tss. It is suggested that the I/O module save its state to fname.ios. Note that any events placed in the event queue by the I/O module will be saved (and restored) by GRMON.

```
void (*save)(char *fname);/* save simulation state */
```

The restore() function is called when restore command is issued in the simulator. The I/O module should restore any required state to resume operation from a saved check-point. *fname points to the base file name which is used by the simulator backend - GRMON restores the simulator internal state from *fname.tss.*

6.4.1.4 Cygwin specific io_init()

Due to problems of resolving cross-referenced symbols in the module loading when using Cygwin, the io_init() routine in the I/O module must initialise a local copy of simif and ioif. This is done by providing the following io_init() routine:

```
static void io_init(struct sim_interface sif, struct io_interface iif)
{
#ifdef __CYGWIN32__
/* Do not remove, needed when compiling on Cygwin! */
    simif = sif;
    ioif = iif;
#endif
/* additional init code goes here */
};
```

6.4.2 LEON AHB emulation interface

In addition to the above described I/O modules, GRMON also allows emulation of the LEON processor core with a completely user-defined memory and I/O architecture. By loading an AHB module (ahb.so), the internal memory emulation is disabled. The emulated processor core communicates with the AHB module using an interface similar to the AHB master interface in proc.vhd of the LEON VHDL model. The AHB module can then emulate the complete AHB bus and all attached units.

The AHB module interface is made up of two parts; one that is exported by GRMON and defines GRMON functions and data structures that can be used by the AHB module; and one that is exported by the AHB module and allows GRMON to access the emulated AHB devices.

At start-up, GRMON searches for 'ahb.so' in the current directory, but the location of the module can be specified using the **-ahbm** switch. Note that the module must be compiled to be position-independent, i.e. with the -fPIC switch (gcc).

6.4.2.1 procif structure

GRMON exports one structure for AHB emulation: procif. The procif structure defines a few functions giving access to the processor emulation and cache behaviour. The procif structure is defined in tsim.h:

```
struct proc_interface {
    void (*set_irl)(int level); /* generate external interrupt */
    void (*cache_snoop)(uint32 addr);
    void (*cctrl)(uint32 *data, uint32 read);
    void (*power_down)();
};
extern struct proc_interface procif;
```

The elements in the structure have the following meaning:

```
void (*set_irl)(int level);
```

Set the current interrupt level (iui.irl in VHDL model). Allowed values are 0 - 15, with 0 meaning no pending interrupt. Once the interrupt level is set, it will remain until it is changed by a new call to set_irl(). The modules interrupt callback routine should typically reset the interrupt level to avoid new interrupts.

```
void (*cache_snoop)(uint32 addr);
```

The cache_snoop() function emulates the data cache snooping of the processor. The tags to the given address will be checked, and if a match is detected the corresponding cacheline will be flushed (= the tag will be cleared).

```
void (*cctrl)(uint32 *data, uint32 read);
```

Read and write the cache control register (CCR). The CCR is attached to the APB bus in the VHDL model, and this function can be called by the AHB module to read and write the register. If read = 1, the CCR value is returned in *data, else the value of *data is written to the CCR.

```
void (*power_down)();
```

Performs a system reset. Should only be used if the I/O device is capable of driving the reset input.

6.4.2.2 Structure to be provided by AHB module

tsim.h defines the structure to be provided by the emulated AHB module:

```
struct ahb_access {
    uint32 address;
    uint32 *data;
    uint32 ws;
    uint32 rnum;
    uint32 wsize;
    uint32 cache;
};
struct ahb_subsystem {
    void (*init)();/* called once on start-up */
    void (*exit)();/* called once on exit */
    void (*reset)();/* called on processor reset */
    void (*restart)();/* called on simulator restart */
    int (*read)(struct ahb_access *access);
    int (*write)(struct ahb_access *access);
    int (*diag_read)(uint32 addr, int *data);
    int (*diag_write)(uint32 addr, int *data);
    char *(*get_io_ptr)(unsigned int addr, int size);
    int (*command)(char * cmd); /* I/O specific commands */
    void (*save)(char * fname); /* save state */
    void (*restore)(char * fname); /* restore state */
    int (*intack)(int level); /* interrupt acknowledge */
};
```

The elements of the structure have the following meanings:

```
void (*init)();
```

Called once on simulator startup. Set to NULL if unused.

```
void (*exit)();
```

Called once on simulator exit. Set to NULL if unused.

```
void (*reset)();
```

Called every time the processor is reset (i.e also startup). Set to NULL if unused.

```
void (*restart)();
```

Called every time the simulator is restarted (simtime set to zero). Set to NULL if unused.

```
int (*read)(struct ahb_access *ahbacc);
```

Processor AHB read. The processor always reads one or more 32-bit words from the AHB bus. The ahb_access structure contains the access parameters: access.addr = read address; access.data = pointer to the first read data; access.ws = should return the number of AHB waitstates used for the complete access; access.rnum = number of words read (1 - 8); access.wsize = not used during read cycles; access.cache = should return 1 if the access is cacheable, else 0. Return values: 0 = access succeeded; 1 = access failed, generate memory exception; -1 = undecoded area, continue to decode address (I/O module or LEON registers).

```
int (*write)(struct ahb_access *ahbacc);
```

Processor AHB write. The processor can write 1, 2, 4 or 8 bytes per access. The access parameters are as for read() with the following changes: access.data = pointer to first write data; access.rnum = not used; acess.wsize = defines write size as follows: 0 = byte, 1 = half-word, 2 = word, 3 = double-word. Return values as for read()

```
char * (*get_io_ptr)(unsigned int addr, int size);
```

During file load operations and displaying of memory contents, GRMON will access emulated memory through a memory pointer. get_io_ptr() is called with the target address and transfer size, and should return a character pointer to the emulated memory array if the address and size is within the range of the emulated memory. If outside the range, -1 should be returned. Set to NULL if not used

.

```
int (*command)(char * cmd);
```

The AHB module can optionally receive command-line commands. A command is first sent to the AHB and I/O modules, and if not recognised, the GRMON. command() is called with the full command string in *cmd. Should return 1 if the command is recognized, otherwise 0. When GRMON commands are issued through the gdb 'monitor' command, a return value of 0 or 1 will result in an 'OK' response to the gdb command. A return value > 1 will send the value itself as the gdb response. A return value < 1 will truncate the lsb 8 bits and send them back as a gdb error response: 'E*nn*'.

```
void (*save)(char *fname);/* save simulation state */
```

The save() function is called when save command is issued in the simulator. The AHB module should save any required state which is needed to completely restore the state at a later stage. *fname points to the base file name which is used by GRMON - GRMON saves the internal state of the simulator backend to xi. It is suggested that the AHB module save its state to fname.ahs. Note that any events placed in the event queue by the AHB module will be saved (and restored) by GRMON.

```
void (*save)(char *fname);/* save simulation state */
```

The restore() function is called when restore command is issued in the simulator. The AHB module should restore any required state to resume operation from a saved check-point. *fname points to the base file name which is used by GRMON - GRMON restores the internal state of the simulator backend from fname.tss.

```
int (*intack)(int level);
```

intack() is called when the processor takes an interrupt trap (tt = 0x11 - 0x1f). The level of the taken interrupt is passed in level. This callback can be used to implement interrupt controllers. intack() should return 1 if the interrupt acknowledgement was handeled by the AHB module, otherwise 0. If 0 is returned, the default LEON interrupt controller will receive the intack instead.

6.4.3 Co-processor emulation

6.4.3.1 FPU/CP interface

The professional version of GRMON can emulate a user-defined floating-point unit (FPU) and co-processor (CP). The FPU and CP are included into the simulator using loadable modules. To access the module, the structure 'cp_interface' defined in io.h. The structure contains a number of functions and variables that must be provided by the emulated FPU/CP:

```
/* structure of function to be provided by an external co-processor */
struct cp_interface {
    void (*cp_init)();               /* called once on start-up */
    void (*cp_exit)();               /* called once on exit */
    void (*cp_reset)();              /* called on processor reset */
    void (*cp_restart)();            /* called on simulator restart */
    uint32 (*cp_reg)(int reg, uint32 data, int read);
    int (*cp_load)(int reg, uint32 data, int *hold);
    int (*cp_store)(int reg, uint32 *data, int *hold);
    int (*cp_exec)(uint32 pc, uint32 inst, int *hold);
    int (*cp_cc)(int *cc, int *hold); /* get condition codes */
    int *cp_status;                  /* unit status */
    void (*cp_print)();              /* print registers */
    int (*command)(char * cmd);      /* CP specific commands */
};
```

6.4.3.2 Structure elements

```
void (*cp_init)();
```

Called once on simulator startup. Set to NULL if not used.

```
void (*cp_exit)();
```

Called once on simulator exit. Set to NULL if not used.

```
void (*cp_reset)();
```

Called every time the processor is reset. Set to NULL if not used.

```
void (*cp_restart)();
```

Called every time the simulator is restarted. Set to NULL if not used.

```
uint32 (*cp_reg)(int reg, uint32 data, int read);
```

Used by the simulator to perform diagnostics read and write to the FPU/CP registers. Calling cp_reg() should not have any side-effects on the FPU/CP status. 'reg' indicates which register to access: 0-31 indicates %f0-%f31/%c0- %31, 34 indicates %fsr/%csr. 'read' indicates read or write access: read==0 indicates write access, read!=0 indicates read access. Written data is passed in 'data', the return value contains the read value on read accesses.

```
int (*cp_load)(int reg, uint32 data, int *hold);
```

Used to perform FPU/CP load instructions. 'reg' indicates which register to access: 0-31 indicates %f0-%f31/ %c0- %31, 34 indicates %fsr/%csr. Loaded data is passed in 'data'. If data dependency is emulated, the number of stall cycles should be return in *hold. The return value should be zero if no trap occurred or the trap number if a trap did occur (0x8 for the FPU, 0x28 for CP). A trap can occur if the FPU/CP is in exception_pending mode when the load is executed.

```
int (*cp_store)(int reg, uint32 *data, int *hold);
```

Used to perform FPU/CP store instructions. 'reg' indicates which register to access: 0-31 indicates %f0-%f31/ %c0- %31, 32 indicates %fq/%cq, 34 indicates %fsr/%csr. Stored should be assigned to *data. During a STDFQ, the %pc should be assigned to data[0] while the instruction opcode to data[1]. If data dependency is emulated, the number of stall cycles should be return in *hold. The return value should be zero if no trap occurred or the trap number if a trap did occur (0x8 for the FPU, 0x28 for CP). A trap can occur if the FPU/CP is in exception_pending mode when the store is executed.

```
int (*cp_exec)(uint32 pc, uint32 inst, int *hold);
```

Execute FPU/CP instruction. The %pc is passed in 'pc' and the instruction opcode in 'inst'. If data dependency is emulated, the number of stall cycles should be return in *hold. The return value should be zero if no trap occurred or the trap number if a trap did occur (0x8 for the FPU, 0x28 for CP). A trap can occur if the FPU/CP is in exception_pending mode when a new FPU/CP instruction is executed.

```
int (*cp_cc)(int *cc, int *hold); /* get condition codes */
```

Read condition codes. Used by FBCC/CBCC instructions. The condition codes (0 - 3) should be returned in *cc. If data dependency is emulated, the number of stall cycles should be return in *hold. The return value should be zero if no trap occurred or the trap number if a trap did occur (0x8 for the FPU, 0x28 for CP). A trap can occur if the FPU/CP is in exception_pending mode when a FBCC/CBCC instruction is executed.

```
int *cp_status;/* unit status */
```

Should contain the FPU/CP execution status: 0 = execute_mode, 1 = exception_pending, 2 = exception_mode.

```
void (*cp_print)();/* print registers */
```

Should print the FPU/CP registers to stdio.

```
int (*command)(char * cmd); /* CP specific commands */
```

User defined FPU/CP control commands. NOT YET IMPLEMENTED.

6.4.3.3 Attaching the FPU and CP.

At startup the simulator tries to load two dynamic link libraries containing an external FPU or CP. The simulator looks for the file fp.so and cp.so in the current directory and in the search path defined by ldconfig. The location of the modules can also be defined using **-cpm** and **-fpm** switches. Each library is searched for a pointer 'cp' that points to a cp_interface structure describing the co- processor. Below is an example from fp.c:

```
struct cp_interface test_fpu = {
```

```
    cp_init, /* cp_init */
    NULL, /* cp_exit */
    cp_init, /* cp_reset */
    cp_init, /* cp_restart */
    cp_reg, /* cp_reg */
    cp_load, /* cp_load */
    cp_store, /* cp_store */
    fpmeiko, /* cp_exec */
    cp_cc, /* cp_cc */
    &fpregs.fpstate,/* cp_status */
    cp_print, /* cp_print */
NULL    /* cp_command */
};
struct cp_interface *cp = &test_fpu; /* Attach pointer!! */
```

6.4.3.4 Example FPU

The file fp.c contains a complete SPARC FPU using the co-processor interface. It can be used as a template for implementation of other co-processors. Note that data-dependency checking for correct timing is not implemented in this version (it is however implemented in the built-in version of TSIM).

6.5 Limitations

On Windows platforms GRMON simulator backend is not capable of reading UART A/B from the console, only writing is possible. If reading of UART A/B is necessary, the simulator should be started with -nouart, and emulation of the UARTs should be handled by the I/O module.

APPENDIX A: HASP

A.1 Installing HASP Device Driver

A.1.1 On a Windows NT/2000/XP Station

The HASP device driver is installed automatically when using the HDD32.EXE Win32 software setup. You'll find this applications in the HASP drivers directory of your GRMON CD. It automatically recognize the operating system in use and install the correct driver files at the required location.

Note: To install the HASP device driver under Windows NT/2000/XP, you need administrator privileges.

A.1.2 On a Linux platform

The HASP software for Linux includes the following:

• Kernel mode drivers for various kernel versions
• Utilities to query the driver version and to display parallel ports
• HASP library

It is contained in the *redhat-1.05-1.i386.tar.gz*, *suse-1.5-1.i386.tar.gz* or the *haspdriver.tar.gz* archive in the Linux directory on the GRMON CD. For detailed information on the components refer to the readme files in the archive.

Note: All described action should be executed as root.

Aladdin Daemon Installation (*aksusbd*)

Enabling Access to USB Keys

In order for the daemon to access USB keys, the so-called *usbdevfs* must be mounted on */proc/bus/usb*. On newer distributions it is mounted automatically (e.g SuSe 7.0). To mount *usbdevfs* manually use the following command:

```
mount -t usbdevfs none /proc/bus/usb
```

Enabling Access to Parallel Keys

To enable access to parallel port keys, the kernel driver *aksparlnx* must be installed before starting *aksusbd*.

Loading the Daemon

Load the daemon by starting it:

```
<path>/aksusbd
```

The daemon will fork and put itself into the background.

The status message is generated in the system log informing you if the installation has been successful or not. It reports its version, the version of the API used for USB and the version of the API inside the kernel driver (for parallel port keys).
If the kernel driver happens to be unavailable when *aksusbd* is launched, parallel port keys cannot be accessed, but USB keys are still accessible. The system log reflects this status.

If */proc/bus/usb* is not mounted when launching *aksusbd*, USB keys cannot be accessed.

Preferably the daemon should be started at system boot up time with some script located in */etc/rc.d/init.d* or */etc/init.d* (depending on Linux distribution).

Command Line Switches for *aksusbd* (Linux)

-v	Print version number as decimal, format xx.xx.
-l <*value***>**	Select type of diagnostic messages. Possible values are: 0 - only errors, 1- normal (default), 2 - verbose, 3 - ultra verbose. The messages are logged in syslog with priority kern.info (and kern.debug). Refer to */etc/syslog.conf* to see where the messages will be put, usually it is the file */var/log/* messages.
-u <*value***>**	Specifies the permission bits for the socket special file. Default is 666 (access for everyone).
-h	Print command line help

A.2 Installing HASP4Net License Manager

The following steps are necessary to install HASP4 Net in a network:

• Install the appropriate HASP device driver or daemon and connect the HASP4 Net key.
• Install and start the HASP License Manager on the same machine.
• Customize the HASP License Manager and the HASP4 Net client, if necessary.

A.2.1 On a Windows NT/2000/XP Station

The HASP License Manager for Windows NT/2000/XP is *nhsrvice32.exe*. Use the setup file *lmsetup.exe* to install it. It is recommended that you install the HASP License Manager as an NT service, so there is no need to log in to the station to provide the functionality.

1. Install the HASP device driver and connect the HASP4 Net key to a station.

2. Install the License Manager by running lmsetup.exe from your GRMON CD and following the instructions of the installation wizard. As installation type, select Service.

To activate the HASP License Manager application, start it from the Start menu or Windows Explorer. The HASP License Manager application is always active when any protocol is loaded and a HASP4 Net key is con ected. To deactivate it, select Exit from the main menu.

A.2.2 On Linux station

Before installing the LM you must install the HASP driver and *aksusbd* daemon. Follow the installation instructions in the README file provided in the "linux" directory.

If you're using SuSE 7.3 or 8.0, you can install the following SuSE RPM package:

```
rpm -i hasplm-suse-8.08-1.i386.rpm
```

It you're using RedHat 7.2 or 7.3, you can install the following RedHat RPM package:

```
rpm -i hasplm-redhat-8.08-1.i386.rpm
```

It you're running a different Linux distribution, you must install the HASP LM manually:

Unpack the archive using

```
tar -xzf [path/]linux-hasplm_8_08.tar.gz
```

This will create a "linux-hasplm_8_08" directory.

Change into this directory and execute as root

```
./dinst
```

This will install the LM and arrange the system startup scripts so that the LM will automatically start at system boot.

If you have any firewall software installed, please make sure that traffic to/from port 475/udp is permitted. Depending on the Linux version you can use the ipfwadm or ipchains utilities to query/change the filrewall settings.

APPENDIX B: GRMON Command description

Command	Target	Description
ahb [trace_length]	dsu	Show AHB trace.
aprof [0|1] [period]	sim	enable/disable accumulative profiling. No arguments shows the collected profiling statistics.
batch [-echo] <batchfile>	dsu/sim	Execute a batch file of grmon commands from <batchfile>. Echo commands if -echo is specified.
baud <rate>	dsu	Change DSU baud rate.
break [addr]	dsu/sim	Print breakpoints or add breakpoint if addr is supplied. Text symbols can be used instead of an address.
bt	sim	Print backtrace.
cont	dsu/sim	Continue execution.
cp	sim	Show registers in co-processor (if present).
dcache	dsu/sim	Show data cache.
debug [level]	dsu/sim	Change or show debug level.
delete <bp>	dsu/sim	Delete breakpoint ‘bp’.
disassemble [addr [cnt]]	dsu/sim	Disassemble [cnt] instructions at [addr].
echo	dsu/sim	Echo string in monitor window.
exit	dsu/sim	Alias for ‘quit’, exits monitor.
flash	dsu	Print the detected flash memory configuration.
flash disable	dsu	Disable writes to flash memory.
flash enable	dsu	Enable writes to flash memory.
flash erase [addr]|all	dsu	Erase flash memory blocks.
flash lock [addr]|all	dsu	Lock flash memory blocks.
flash lockdown [addr]|all	dsu	Lockdown flash memory blocks.
flash query	dsu	Print the flash memory query register contents.
flash status	dsu	Print the flash memory block lock status.
flash unlock [addr]|all	dsu	Unlock flash memory blocks.
flash write [addr] [data]	dsu	Write single data value to flash address.
float	dsu/sim	Display FPU registers.

Table 1: GRMON Commands

Command	Target	Description
gdb	dsu/sim	Connect to the GNU debugger (gdb).
go	dsu/sim	Start execution at <addr> without initialisation.
hbreak [addr]	dsu/sim	Print breakpoints or add hardware breakpoint.
help [cmd]	dsu/sim	Show available commands or usage for specific command.
hist [trace_length]	dsu/sim	Show trace history. In sim-mode, this command also enables the instruction tracing. In dsu-mode the tm command is
icache	dsu/sim	Show instruction cache
init	dsu	re-initialise the processor.
inst [trace_length]	dsu	Show traced instructions.
leon	dsu/sim	Show LEON registers.
load <file>	dsu/sim	Load a file into memory. The file should be in ELF32 format.
mem [addr] [count]	dsu/sim	Alias for "x", examine memory. Examine memory at at [addr] for [count] bytes.
mmu	dsu	Print mmu registers.
perf [reset]	sim	show/reset performance statistics
profile [0|1]	dsu/sim	enable/disable simple profiling. No arguments shows the collected profiling statistics.
register [reg|win] [val]	dsu/sim	Show/set integer registers (or windows, eg 're w2')
reset	dsu/sim	Reset the active backend.
restore	sim	Restore simulator state from file.
run	dsu/sim	Run loaded application.
save file	sim	Save simulator state to file.
shell <command>	dsu/sim	Execute a shell command.
stack <addr>	dsu	Set stack pointer for next run.
step [n]	dsu/sim	Single step one or [n] times.
symbols [symbol_file]	dsu/sim	Show symbols or load symbols from file.
target [sim|dsu] [args]	dsu/sim	Change backend (no argument cycles through available backends). '*args*' are arguments passed to the backen to which grmon is switching. If no args is supplied, the switches that were supplied when starting the current backend are reused. Note that only backend specific commands can be supplied to target.
tm [ahb|cpu|both]	dsu	Select trace mode.

Table 1: GRMON Commands

Command	Target	Description
tra [inst_count]	sim	Trace [inst_count] instructions.
quit	dsu/sim	Exit grmon and return to invoker(the shell).
va <addr>	dsu	Performs a virtual-to-physical translation of address.
verify <file>	dsu	Verify memory contents against file.
version	dsu/sim	Show version.
watch [addr]	dsu/sim	Print or add watchpoint.
wmem <addr> <data>	dsu/sim	Write <data> to memory at address <addr>.
x [addr] [count]	dsu/sim	Examine memory at at [addr] for [count] bytes.

Table 1: GRMON Commands